VNF Performance Modelling: from Stand-alone to Chained Topologies.

Steven Van Rossem, Wouter Tavernier, Didier Colle, Mario Pickavet, Piet Demeester

Ghent University - imec, IDLab, Technologiepark-Zwijnaarde 126, B-9052 Ghent, Belgium

Abstract

One of the main incentives for deploying network functions on a virtualized or cloud-based infrastructure, is the ability for on-demand orchestration and elastic resource scaling following the workload demand. This can also be combined with a multi-party service creation cycle: the service provider sources various network functions from different vendors or developers, and combines them into a modular network service. This way, multiple virtual network functions (VNFs) are connected into more complex topologies called service chains. Deployment speed is important here, and it is therefore beneficial if the service provider can limit extra validation testing of the combined service chain, and rely on the provided profiling results of the supplied single VNFs. Our research shows that it is however not always evident to accurately predict the performance of a total service chain, from the isolated benchmark or profiling tests of its discrete network functions. To mitigate this, we propose a two-step deployment workflow: First, a general trend estimation for the chain performance is derived from the stand-alone VNF profiling results, together with an initial resource allocation. This information then optimizes the second phase, where online monitored data of the service chain is used to quickly adjust the estimated performance model where needed. Our tests show that this can lead to a more efficient VNF chain deployment, needing less scaling iterations to meet the chain performance specification, while avoiding the need for a complete proactive and time-consuming VNF chain validation.

Keywords: Network Function Virtualization (NFV), DevOps, NFV Performance Profiling, Network function chain deployment

1. Introduction

With the advent of cloud and edge computing, an unseen deployment flexibility for softwarized communication services is enabled. From the increasingly maturing field of Network Function Virtualization (NFV), more and more implementations of Virtualized Network Functions (VNFs) such as routers, load-balancers or firewalls are becoming available. The expected service performance is specified by a number of Key Performance

*Corresponding author

Email address: stevenvanrossem@gmail.com (Steven Van Rossem)

Indicators (KPIs) with agreed limits in the Service Level Agreement (SLA). The SLA is then the contract between a service provider and its customers, the service users. Typical KPI metrics are for example packet loss, response time or processing latency. The performance of these VNFs is determined by the amount of resources they are allowed consume in the cloud-based infrastructure where they are running. An example of such a resource is virtualized CPU processing power (vCPU). This can be a number of multi-threaded CPU cores, or a percentage of one or more physical CPUs. Other resources which can be virtualized are for example memory, storage or network bandwidth. NFV technology allows virtual resources to be allocated on-demand, growing or decreasing together with the workload demand. From a cost saving perspective, the service provider can tune the amount of allocated resources, as long as the users do not experience service degradation, specified in the SLA. But to efficiently control the service quality, the relation between allocated resources and resulting service performance should be known. We define this relation as the service performance profile. If such profile does not exists, the service provider must stepwise modify resource allocations, until performance is met. This more like a trial-and-error method, which can take multiple iterations before an optimal point is found. The performance profile must now help the service provider to estimate an optimal amount of resources to allocate, in order to meet the SLA under the current workload. If the workload is known or can be pro-actively predicted, then an adequate resource allocation can be estimated, resulting in less over- or under-provisioning.

In this paper we focus specifically on the performance profile of service chains, where network traffic is traversing multiple VNFs. The research goal can be formulated as:

"Starting from a limited number of VNFs, whose performance profiles are known in advance through profiling, how accurate can we predict the chained performance when those VNFs are combined."

When VNFs are tested outside of the chain, we refer to them as stand-alone VNF profiles. The relevance of this research question is justified by two main observations:

- The service chain creation process is naturally based on stand-alone VNF profiling.
- The test time of service chains can be greatly reduced if we can rely on the performance profiles of the stand-alone VNFs only.

The above statements are explained more in detail in the following subsections.

1.1. The Service Chain Creation Process

Modern, DevOps-based, software development is also being adopted by the telecom industry [1]. From a testing perspective this means that VNFs are first tested in a stand-alone way, before they are handed over to the service provider for integration into the service chain. This is displayed in Fig. 1. When a service chain is created, the stand-alone VNF profile data is therefore likely to be already available. The usability of those stand-alone profiles is further increased by the use of a shared test infrastructure:

- Using cloud native design principles [2] [3], a virtualized network service can leverage a shared Infrastructure as a Service (IaaS). The IaaS can be used both for VNF

testing as production environment. This enables a very representative staging environment for accurate VNF performance measurements and a base for multi-party service creation [1] [4].

- The service provider can source multiple VNF implementations from external suppliers, and then combine those VNFs himself into a custom chain. This enables an eco-system in which performance validation can be automated and delegated to multiple VNF suppliers, before acceptance by the service provider [5]. On a broader scale this also leads to new ways of cooperation between service providers and VNF developers [6] [7], where service performance management is a continuous joint process between developers and operators [8].

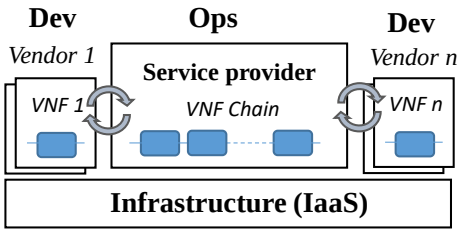


Figure 1: The development of a VNF chain is based on the stand-alone testing of the discrete VNFs by multiple Vendors.

The above principles are illustrated in Fig. 1, where the service provider creates a custom service chain, in cooperation with multiple VNF suppliers (Vendors). To alleviate the validation effort of the service provider, it makes sense to align the Dev testing at the Vendors' side as much as possible with the Ops environment. The use of a shared IaaS environment can certainly help here. In the remainder of this paper, we assume that stand-alone VNF profiling is done using a workload and environment aligned with the operational context of the service provider.

1.2. Gains in Profiling Time

In the previous subsection 1.1, we explained how the service provider can combine existing VNFs into new service chains. However, due to time and resource constraints, it is not always possible to fully validate the new VNF chain with every possible workload and resource allocation combination; e.g. time to market is urgent, critical updates like a security patch must be quickly introduced or not enough resources are available in the IaaS to completely duplicate the VNF chain for testing. Therefore it would be convenient to derive a chained performance model, from the stand-alone VNF models. The difference in total test effort can be simply shown by the example in Fig. 2. Let us assume each VNF has n possible deployment combinations to test. Each combination can be an available resource flavor (e.g. reserved number of vCPUs and network bandwidth). If each VNF is tested stand-alone (Fig. 2 a), the total number of combinations is additive. When the VNFs are chained (Fig. 2 b), the total number of combinations is multiplicative. The graph depicts the difference in test time between the two approaches. This simple example motivates to investigate the possible gain in test effort, if we could use the test results from the stand-alone VNFs to predict the performance of the chained VNFs.

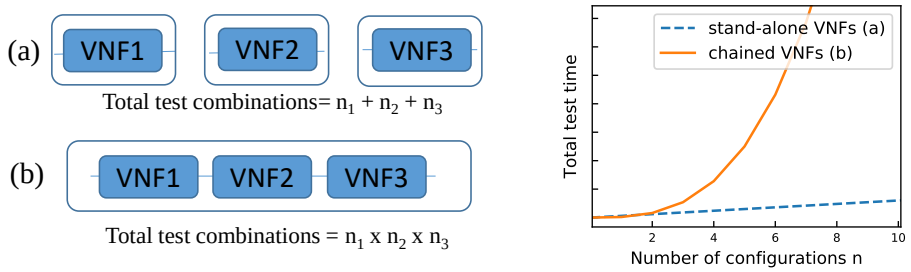


Figure 2: (a) Each VNF is tested stand-alone. (b) The VNF chain is tested as a whole.
The total test time grows exponentially when considering all VNF chain combinations (illustrated for 3 VNFs).

1.3. Challenges in VNF Chain Performance Modelling

We start by profiling the performance of a limited number of stand-alone VNFs, using similar techniques as in [9]. We extend this work by focusing specifically on the performance profile of service chains, consisting of multiple VNFs. We combine the tested VNFs into service chains and verify if the combined performance is accurately predictable from the stand-alone tests only. A logical assumption is that the performance of the VNF chain matches the performance of the weakest link in the chain. The main contribution of this paper lies in the investigation of the above assumption: Since we see large prediction errors, we analyze why the prediction of chained VNF performance cannot rely solely on the stand-alone VNF profiles. We further propose a two-phased profiling workflow to tackle this, as introduced in Fig. 3.

The outline of the paper is as follows. In Section 3 we explain the mathematical models and notations which are used to create a performance profile for the total VNF chain. Next in Section 3, we briefly introduce the VNFs used to test and validate our modelling approaches. A first test is conducted in Section 4, where we see that the stand-alone VNF profiles can bootstrap the resource allocation of a VNF chain. This results in a faster and more resource efficient service chain orchestration. We continue in Section 5 to validate if the stand-alone VNF profiles can be generally used to predict the VNF chain performance for all possible configurations of workloads and resource allocations of the chained VNFs. We see that care needs to be taken, since the chained VNF performance cannot always be accurately predicted from the stand-alone profiles only. This is more deeply investigated in Section 6, where a technical analysis is done on the earlier anomalous VNF chain prediction results. Finally, in Section 7, we propose a workflow to mitigate possible prediction errors, this online adjustment procedure is summarized in Fig. 3. In Phase 1, the profiled data from the stand-alone VNFs is used to model an initially estimated performance trend of the chained service (as done in the Dev environments in Fig. 1). When the actual chain is deployed in Phase 2, the selected model is transferred and further re-trained with online gathered data (as done in the Ops environment in Fig. 1).

Using this approach, we investigate which modelling method needs a minimal range of additional training data in Phase 2. Using this two-phased workflow, we can bootstrap the resource provisioning for a VNF chain, using only data from stand-alone VNF

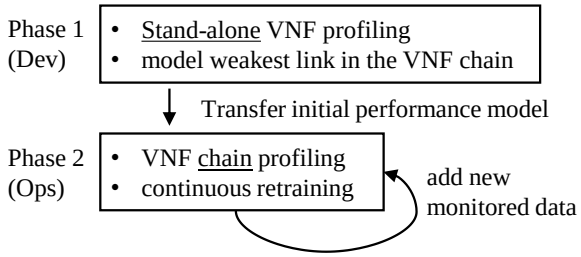


Figure 3: Overview of the presented profiling workflow.

profiling. To add reliability that the estimated resource allocation will certainly achieve the intended VNF chain performance, the second phase is added. In total, this approach is more time efficient than exhaustively testing all possible combinations as in Fig. 2. Finally, we discuss the used modelling approach in Section 8.1, where we highlight several learnings and ideas for further research.

2. Related Work

Our work builds further on previous learnings in the field of VNF profiling and the analysis of the resulting data. In the following paragraphs we highlight publications which have reported similar performance measurements of profiled VNFs. In general, our paper goes further by investigating deeper the predictability and characterization of VNF performance trends.

Our experiments relate to work done in [10] and [11], where the performance of a VNF chain is investigated and related to the stand-alone VNFs. There it is shown that VNFs can behave differently in a chained topology versus stand-alone, and also when the order of the VNF chain is altered. In general, our tests also confirm this. Both papers conclude that VNF chain performance is hard to predict from isolated VNF performances, and prediction errors are likely. The authors in [11] provide a measurement campaign to show this, but without any root cause investigation or mitigation proposal. The preliminary conclusion is that chained VNF performance can only be accurately predicted, if the chain is profiled as a whole. Our work on the other hand, tries to mitigate the predicted chain performance by online model retraining.

In [10], a deeper investigation is done why the performance of a single VNF changes, when placed in a VNF chain. The investigated VNF chain performance degradation seems caused by network I/O bottlenecks in the underlying infrastructure, which is dynamically influenced if multiple VNFs are competing for the same network I/O resources. We try to avoid this in our tests by carefully isolating all the used VNFs and traffic generating functions to separated vCPU cores. We investigate this further in Section 6 and also unmask several TCP optimization mechanisms in the Linux kernel as manipulators of the chained VNF performance. We additionally propose a hybrid approach to mitigate VNF chain prediction, in which stand-alone VNF profiling still provides useful information, used to optimize the prediction of the VNF chain performance, both in profiling time as in accuracy. We executed measurements on multiple VNF chains to validate this more broadly.

An online adjustment of a VNF performance model is presented in [12] and [13], based on analytical curve fitting. The reported test cases compromise however only single dimensional workload spaces and monitor the service only directly in production, without any prior profiling in a development environment. The methods investigated in our paper, can be easily extended to multi-variate input spaces for both workload and resource configuration parameters. Moreover, we propose a proactive profiling phase to determine an initial model which is adjusted in production afterwards.

In our tests, the VNFs are connected using normal TCP based links, forwarded through standard Linux bridges in the kernel, this default network layer is not modified. There is however much room for optimization in the Linux network layer. IPv6 Segment Routing (SRv6) is for example a promising solution to support advanced services which use Service Function Chaining, similar to the VNF chains in our tests. A profiling framework for this routing architecture is presented in [14]. The research in this work shows that not only the VNFs themselves are influencing the chain behaviour, also the data plane connecting the VNFs (implemented in the Linux kernel or underlying network infrastructure), can have a determining effect on the overall chain performance. A comparison is made between the performance of the Linux kernel implementation and the VPP software router for service chaining on the traffic forwarding level. This related work complements the analysis in Section 6, where the network layer in the Linux kernel is found to influence the overall VNF chain performance.

Modern communication networks require flexible, fast and reliable service deployments [15]. Together with the growing availability of cloud native deployment possibilities [3], a trade-off between flexibility and deterministic performance emerges. Adequate profiling can help to model this dynamic service performance, and this could be used as input to pro-actively predict service topologies' performances [4] [16]. The referred work in the following paragraphs exemplifies how VNF profiling, and the resulting VNF performance model, could alleviate the VNF orchestration process.

In [17, 18, 19] it is illustrated how knowledge of the VNF resource usage can optimize VNF chain deployment and calculate the VNF placement across distributed server nodes. In [17] the CPU consumption of an edge router device is profiled in function of multiple parameters of the VNF chain traffic (e.g. packet size, rate, chain length). The profiled router performance trends are used as input in an Integer Linear Programming (ILP) problem to find the optimal deployment of a VNF chain across edge devices. In [18, 19] also a cost function for the resource utilization is derived and minimized using proposed algorithms. Our paper could further complement the cost minimization problem, by providing a pro-active estimation of the needed resources of the total VNF chain. Similarly in [20], a set of profiled, container based VNFs is proven to be light-weight enough to be deployed on a variety of resource-limited edge devices. The work shows how profiled VNF data can alleviate VNF chain orchestration. The profiled performance trends are however visually analysed for a single example use case only, so this method might not scale well for other edge devices or VNFs. Our work focuses on more generic modelling methods, which can be automatically applied to other chain topologies also. We mainly highlight the preparatory process of estimating the chain performance trend, before it can be used as input for other VNF orchestration processes.

The work in [21, 22, 23] is a complement to the scaling algorithms used in Section 4. But while the referenced works focus on supervised machine learning to drive the auto-scaling, our approach uses linear regression-based techniques, which prove to need less

training data and provide a better accuracy. A comparison with machine learning-based methods is done in Section 5.2. As an alternative to the retraining of the model in Phase 2 (see Fig. 3), another machine learning method called "reinforcement learning" is used in [23]. As explained in Section 7, machine learning-based methods are not capable to extrapolate the learned VNF performance to untested VNF configurations. Our proposed method can instead use linear regression to do this. This results in fewer training samples needed compared to purely machine learning-based modelling methods.

Another use case for profiled VNF data is anomaly detection in complex VNF topologies. In [24], an online cloud performance debugging system is based on a deep learning model. The input values correspond to network queue depths, and the output values to the probability for a given VNF to initiate a performance violation. To train the model, a large amount of data is used (one week's worth of trace data). Similarly in [8], a binary classifier is trained which predicts if a large VNF topology is operating within saturated resources or not. It is assumed that a large set of pre-labeled training data is gathered beforehand. However no KPI values are considered in the classifier, therefore no quantitative link can be made to the SLA or resource allocation of the VNF chain. To support anomaly detection mechanisms, our work can alleviate the creation of training data by speeding up the profiling of the whole VNF chain in given configurations. Additionally, in our tests, a trained regression model of the performance can predict performance trends better, compared to a classification model. This leads to a better quantitative assessment of SLA specifications.

3. Test Setup and Background

In this section we introduce the test setup and VNFs which were used. We also summarize the mathematical models and notation based on [9] and further used in this paper. In our approach, the service chain is a linked set of modular functional blocks. We consider a VNF as one distinct functional block, which has its own isolated resources. In practice, the VNF is a single Virtual Machine (VM) or container in our tests. The profiling framework we use for automated workload generation and KPI monitoring is described in [25]. While the workload is flowing through the chain, specialized probes then export metrics related to resource usage, workload and performance. The monitored data is recorded for later analysis. The profiled topologies are given below. Figure 4a is the test setup used in Phase 1 of the profiling workflow in Fig. 3, Figure 4b is used in Phase 2.

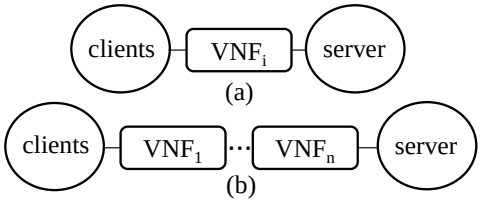


Figure 4: The used test topologies: (a) the stand-alone VNF and (b) chained VNFs.

3.1. VNFs Under Test

The used test hardware are multiple equal compute nodes with 2x 8core Intel E5-2650v2 (2.6GHz) CPU running Ubuntu 18.04 as operating system with the Ubuntu Linux 4.18 kernel. Linux Bridge is used as the hypervisor switch for the VNFs. We do not change default OS options (e.g. we leave hyperthreading enabled). Depending on the virtualization of the VNF (container or VM) we use the configuration options of Docker resp. KVM to isolate the CPU cores between the VNFs under test, the clients and server.

We generate the workload in the clients by continuously generating n concurrent file requests. This means at any given time, there are n clients with each one file request ongoing. *Locust.io* [26] is the tool used to generate this workload. The traffic source is generating file requests of varying filesizes. The traffic sink is a webserver, a Python based implementation (*Flask* [27]) which serves a random file with the requested size back to the clients. All clients request an equally large file. The workload is thus characterized by (i) the number of concurrent clients and (ii) the filesize of the requested files. These are the two workload metrics used in the models.

The VNFs are first profiled stand-alone, meaning that traffic flows only through the single VNF (Fig. 4a). We test the following VNFs:

Haproxy - Load Balancer [28]. An opensource, very fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications. We profile its baseline performance, enabling only round-robin load balancing. The VNF version is v2.0.9, deployed as a Docker container.

Tinyproxy - Proxy Server [29]. An opensource, light-weight HTTP/HTTPS proxy daemon. An ideal solution for use cases such as embedded deployments where a full featured HTTP proxy is required, but the system resources for a larger proxy are unavailable. We profile the baseline default configuration as basic HTTP proxy server. The VNF version is v1.10.0, deployed as a Docker container.

pfSense - Firewall [30]. An opensource firewall and router implementation (that also features unified threat management, load balancing, multi WAN, and more). We profile its baseline performance as traffic forwarder (NAT is enabled) and the default, out-of-the-box firewall configuration. As a side note, we noticed that the TCP offload mechanisms, described in Section 6, are per default disabled for this VNF. The VNF version is v2.4.4-p3, deployed as a VM under KVM. Due to license constraints, only up to one full vCPU can be allocated to this VNF.

Open vSwitch (OvS) - Switch [31]. An opensource production quality, multilayer virtual switch, designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols. For maximum resource isolation, we use the same implementation as in [9]: Open vSwitch v2.10.1 installed in a Virtual Machine based on Alpine Linux v3.9.1. We use the default standalone switch configuration, so a flow entry is inserted for every unique flow passing through, one per unique TCP source-destination port pair. Note that to benefit from multi-core cpu allocation, we must enable multiqueue virtio-net drivers in KVM. This enables packet sending/receiving processing to scale with the number of available vCPUs of the guest.

We must divide the available vCPUs over the specified number of queues in the virtio driver and the processing in the VM (OVS) itself. The VNF is deployed as a VM under KVM.

3.2. Generic Performance Modelling

Our previous work in [9] and [25] focused on profiling the performance of single VNFs. Now we consider the performance of VNF chains, but we can still apply the same performance model if we consider the total VNF chain as a closed system.

$$f(wl, res) = perf \tag{1}$$

where:

wl	=	input workloads (e.g. concurrent requests, filesize)
res	=	resource allocations (e.g. allocated vCPUs for each VNF in the chain)
$perf$	=	service KPI metrics (e.g. response time)

Throughout the paper, we refer to the performance profile of :

- the stand-alone VNF_i (cf. Fig. 4a): $f_i(wl, res) = perf$
- the total chain (cf. Fig. 4b): $f_{chain}(wl, res) = perf$

The performance model f in Eq. 1 predicts the performance at a given workload and resource allocation. In Section 5 we will analyse several methods to obtain f. In order to use the model f to predict a suitable resource allocation, we need to rework Eq. 1 into the following optimization problem:

$$\begin{cases} f(wl, res) & \leq perf_{SLA} \\ wl & \geq wl_{SLA} \\ minimize & cost(res) \end{cases} \tag{2}$$

where:

$cost(res)$	=	a cost function for the resource allocation
wl_{SLA}	=	the targeted workload, specified in the SLA
$perf_{SLA}$	=	the upper performance limit, specified in the SLA

Note that $f(wl, res) \leq perf_{SLA}$ is only generally true for performance metrics which have an upper limit (e.g. maximum loss or response time). When the SLA defines KPIs with lower limits (e.g. minimal throughput or download speed), the opposite inequality sign should be used. To find a solution for Eq. 2, we can use following heuristic, as also proposed in [9]:

$$\forall res_i \in P: \quad \arg\min_{wl} |f(wl, res_i) - perf_{SLA}| = wl_i^{perf} \tag{3}$$

where:

wl_i^{perf}	=	the workload which minimizes the equation, given res_i and $perf_{SLA}$
P	=	the set of all possible resource allocations, ordered by increasing cost
res_i	=	a possible resource allocation

By iterating through the possible resource allocations in order of increasing cost, we find the value of wl_i^{perf}, the workload which yields a performance closest to the SLA target, for a given res_i. A possible strategy to solve Eq. 3 is for example to stepwise increment wl. If we do this for each res_i, eventually the cheapest resource allocation is found which satisfies $wl \geq wl_{SLA}$ in Eq. 2. For an optimal service deployment, we need to find the minimal (cheapest) resource configuration which meets the workload and the performance target (wl_{SLA}, $perf_{SLA}$), specified in the SLA. In a standard profiling workflow (e.g. [32] [25]), we first use monitored data from a profiling test, to train the model f in Eq. 1. Once f is known, we can solve Eq. 2 to predict a suitable resource allocation. It can be seen that the accuracy of Eq. 2 is largely determined by the accuracy of the performance model f. We extend the work in [9] by investigating if we can derive the performance profile of the total VNF chain f_{chain}, from only the stand-alone VNF_i models f_i and a limited amount of extra data.

In our experiments, we use the following practical parameters for the metrics:

workload metrics (wl):

$filesize$	=	the requested file size
concurrent requests	=	the number of simultaneous ongoing file requests

resource metric (res):

$vCPU_i$	=	vCPU allocation of VNF_i tested as stand-alone or in a chain

performance metric ($perf$):

95% *response time*	=	the 95th percentile for response time of the file requests

3.3. Validated Modelling Methods

In this subsection we summarize the different methods we validate later in Section 5 and 7, to model f_i or f_{chain} in our obtained dataset. In the mentioned sections, it will be explained why some methods are preferable to others. The first three methods are able to model complex, non-polynomial and multi-variate relations, and can be considered as the most generic:

Interpolation. Put simply, to find a prediction, this method interpolates between surrounding, profiled samples. The interpolant is constructed by triangulating the input data using Delaunay triangulation, and on each triangle performing linear barycentric interpolation. This method also works multi-dimensional, so we can interpolate between any number of input parameters (wl, res) to predict the performance of an intermediate configuration. We use the method griddata implemented in the Python SciPy library [33]. In [9] and [25], this method provided the best accuracy for profiling stand-alone VNFs. We validate it here again for our VNF chain modelling approach.

Gaussian Process. Following a Bayesian approach to generic regression, a Gaussian Process defines a prior over functions, which can be converted into a posterior over functions using training data. This method is fully documented in [34]. The training and prediction process is however quite calculation intensive, therefore this method becomes less convenient at large sample sizes and high dimensional models. Previous research [25], [35]) shows that Gaussian Processes are well capable of modelling non-polynomial trends of VNF performances. Therefore we also validate it here for our VNF chain use case. The covariance between training samples must be given as a kernel function. For our tests we use: *Constant * (DotProduct + RBF) + WhiteNoise*. This kernel function is capable of modelling approximately linear relations (as we expect from Fig. 7).

Artificial Neural Network (ANN). Neural Networks are typically good at complex function approximation and regression of multi-variate relationships. More complicated trends require however larger layers and nodes in the model, needing more samples to train the model. Here, we test a similar ANN model as used in [9]. We use one single hidden layer with 10 nodes as compromise for a simplified ANN model, easier to train, while still allowing enough fitting capabilities.

The observed VNF performance trends in Section 5.3 suggest however that a more polynomial or even linear relation might be dominant. Therefore we also evaluate a set of linear regression based methods. In order to allow some curvature in the models, we do a polynomial expansion of the input parameters. For example, if the input metric space is two dimensional and of the form $[a, b]$, the degree-2 polynomial expansion is $[1, a, b, a^2, ab, b^2]$. In our tests, the total input space is: $[\textit{filesize}, \textit{concurrent requests}, vCPU_i]$, as explained in the beginning of this section. So the input parameter space is enlarged, offering more fitting possibilities for following linear regression methods:

Lasso. This is a well known regression analysis method that performs both variable selection and regularization to prevent overfitting, in order to enhance the prediction accuracy and interpretability of the model it produces. In our use case, when applying polynomial expansion to the input parameters of the model, the Lasso method is capable to shrink certain coefficients in the linear model up to zero, if that improves the overall accuracy. This is an effective form of feature selection and limiting the terms of the polynomial expansion in the trained model. In the next sections we mention Lasso1, Lasso2 and Lasso3, which represent polynomial expansion of the first, second and third order respectively, before applying the Lasso method.

Elastic Net. This is a linear regression technique related to the Lasso method. While Lasso tries to shrink certain coefficients more aggressively, Elastic Net prefers to keep all coefficients into the model, using a smoother form of coefficient regularization. This leads to less feature selection power (more input terms of the polynomial expansion are kept in the model). Our test show that this approach yields worse accuracy when applied to our datasets. This indicates that in general only few terms of the polynomial expansion are correlated with the measured KPI trends. Similar to Lasso3, we apply polynomial expansion of the third order to Elastic Net3.

Multi Lasso. This customized procedure is based on the method which showed most promising results in [9], used on the profiling of stand-alone VNFs. In our implementation, a Lasso model is trained separately for each profiled configuration, where only the number of concurrent requests is varying as input parameter for the model (and the filesize and resource allocation is kept fixed). A prediction of a new workload value is made by using the previously mentioned Interpolation method between the Lasso models of surrounding nearby configurations. More details of this method are given in [9].

We implemented the above methods using the functions available in the Python based library Scikit-learn [36]. Throughout our validations, we will use several score types to assess the prediction accuracy, and to validate which method works best:

- Mean Average Error (MAE): This is the mean value of the residual errors of the predicted response time.
- Median Average Deviation (MAD): This is the median value of the residual errors of the predicted response time.
- Root Mean Squared Error (RMSE): When the residuals of the predicted values are normally distributed, the RMSE depicts the standard deviation of the residuals. But since MAE > MAD for most of the results, we suspect that the distribution of the residual errors is right skewed and not centralized around the mean. As recommended in [9], we use the RMSE to compare the accuracy of different methods.
- Mean Absolute Percentage Error (MAPE): The errors are expressed as the ratio of the absolute residual error over the actual value.

4. Bootstrapped Service Chain Provisioning

In this section we experimentally validate how the knowledge of the stand-alone profiles f_i, optimizes the resource allocation of a VNF chain. This illustrated by comparing two methods to optimize the resource allocation:

1. Using a greedy algorithm and no profiling info, a VNF chain is iteratively scaled until the SLA is met (Section 4.1).
2. Using the data from pre-profiled stand-alone VNFs, the chain performance and according resource allocation is estimated pro-actively (Section 4.2).

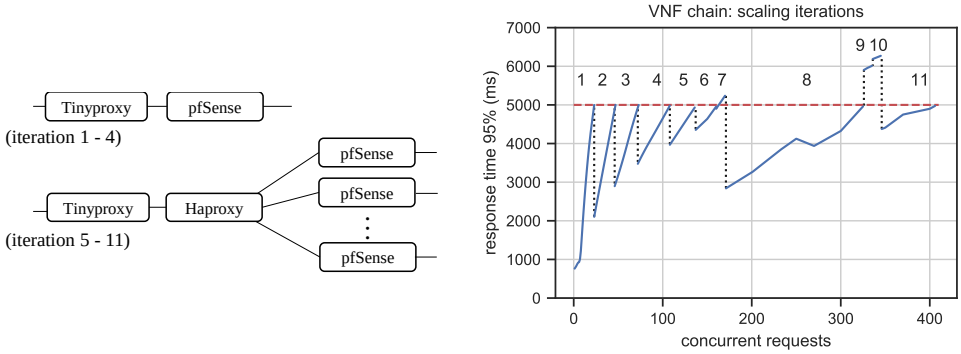
The VNF chain under test is depicted in Fig. 5a. The service functionality is composed out of the proxy features provided by Tinyproxy and the firewall capabilities offered by pfSense. If one of those VNFs becomes the bottleneck, and no more resources are available to allocate, a load balancer (Haproxy) is added and the traffic is balanced over multiple instances of the bottleneck VNF.

For our case study, we want to find the optimal resource allocation to meet following exemplary SLA specification:

- wl_{SLA} = max 400 concurrent requests
- $filesize_{avg}$ = 1MByte average filesize per request
- $perf_{SLA}$ = 95% response time < 5000ms

4.1. Greedy Scaling Algorithm

We first start from the situation where we have no stand-alone VNF profiles f_i available. The workload is gradually increased and extra resources are scaled in whenever the performance limit is reached. We exemplify this with Fig. 5, which shows how eventually, after several resource scaling iterations, the SLA specification is met at the target of 400 concurrent users.



(a) the VNF chain topology

(b) scaling iterations

Figure 5: While the workload is increasing, the VNF chain is scaled using a load balancer (Haproxy), to stay under the KPI (response time) limit.

We follow a greedy procedure (Algorithm 1), which is a generic form of the default, threshold based, scaling procedure implemented in most service platforms such as OpenStack or Kubernetes. When the response time is above the limit, the VNFs which have a vCPU usage above 90% of their allocated value, are scaled and receive a higher vCPU share. We start from a 25% vCPU share for each VNF. The vCPU allocation is incremented in steps of 25% until 1 vCPU (100%) is allocated, afterwards one full vCPU is added per step (100%, 200%, 300%, ...). By using this procedure, we arrive at an appropriate resource allocation after 11 iterations (Fig. 5b):

- Iteration 1-4: Tinyproxy and pfSense start from a vCPU share of 25%. The vCPU shares are iteratively incremented until pfSense becomes the bottleneck.
- Iteration 5: pfSense has reached its max vCPU allocation (100%), from now onwards, Haproxy is added to the chain to load balance the traffic over multiple pfSense instances.
- Iteration 5-10: the number of pfSense instances is incremented from two up to five. Iteratively, also the vCPU shares of Tinyproxy and Haproxy is incremented if they show CPU saturation.
- Iteration 11: the scaling procedure arrives at the specified SLA specification. The final vCPU allocation is: Tinyproxy: 200%, Haproxy: 75%, pfSense: 100% x 5 instances.

Algorithm 1: Greedy scaling procedure

Data: wl_{max}, $perf_{SLA}$
Result: res = chain resource allocation

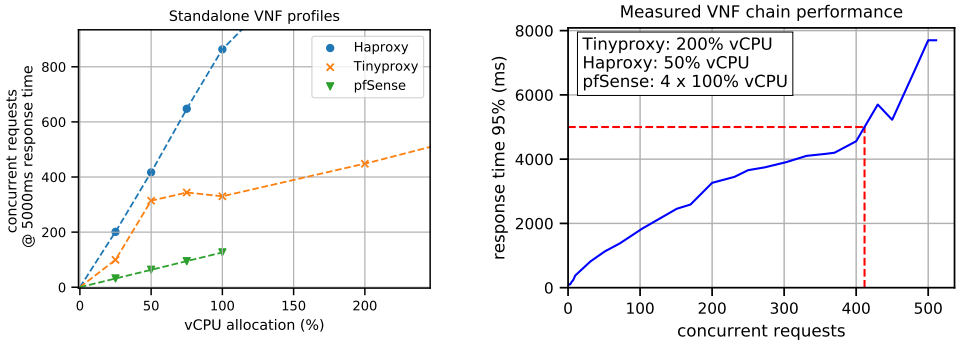
$res \leftarrow$ minimal resources allocated;
$wl \leftarrow$ minimal workload started;
while $wl < wl_{max}$ **do**
  $R \leftarrow$ empty set;
  $KPI \leftarrow$ monitor chain performance;
  foreach $VNF_i \in chain$ **do**
    $res_i \leftarrow$ monitor VNF_i resource usage;
    $R.append(res_i)$;
  end
  if $KPI \geq perf_{SLA}$ **then**
    $res \leftarrow$ `Scale` (res, R) ;
  end
  increase wl;
end
return res;

Function `Scale`(res, R)**:**
  foreach $res_i \in R$ **do**
    if $res_i \geq 90\%$ **then**
      lookup current resource allocation of VNF_i in res ;
      $res \leftarrow$ allocate more resources to VNF_i ;
    end
  end
  return res;

4.2. Resource Allocation Estimation

For comparison with the greedy scaling in previous subsection, we now want to use the profiled data of the stand-alone VNFs (Fig. 5a) to estimate a sufficient resource allocation. We assume that each $VNF_i \in chain$ has been profiled before and a model $f_i(wl, res) = perf$ is available (using the methods selected later in Section 5.2). By using Eq. 2 and 3 we can derive a performance trend for each of the VNFs as shown in Fig. 6a. Each point represents a profiled vCPU allocation. In each point we lookup the maximum workload which could be served within the predefined SLA limit (response time $< 5000ms$, for 1MB file requests). From the graph we can read that to reach 400 concurrent requests, we would need following vCPU allocation: Tinyproxy: 200%, Haproxy: 50%, pfSense: 100% x 4 instances (one pfSense can serve up to 130 concurrent requests). We can see in Fig. 6 the measured performance of this estimated resource allocation. The estimated vCPU allocation is a one-shot iteration, sufficient for the SLA.

In Table 1 we compare again the final resource allocations of the two investigated methods. It is remarkable that the second method, using profiled data, found a more efficient resource allocation to reach the same performance. Moreover, in Fig. 5b we saw that iteration 7, 9 and 10 did not improve the performance. After closer inspection,



(a) stand-alone VNF profiles f_i (filesize = 1MB) (b) f_{chain} performance with SLA limit indication

Figure 6: The predicted resource allocation for the chain meets the required SLA specifications.

	vCPU allocation		
Method	Tinyproxy	Haproxy	pfSense
Greedy	200%	75%	5 x 100%
Profiled	200%	50%	4 x 100%

Table 1: The obtained resource allocation for the tested chain.

we found that these iterations coincide with the scaling of Tinyproxy from 50 to 100% vCPU. In Fig. 6a we can verify that this does not improve the VNF performance a lot. These scaling iterations could be avoided by using the stand-alone profiled data.

4.3. The Need for Further Validation

The previous test indicates that stand-alone VNF profiles can indeed provide a better initial estimation for the needed resource allocation, and thus a faster service provisioning. The stand-alone profiles f_i can be used to bootstrap the orchestration of the VNF chain, by limiting the number of scaling iterations, needed to reach the performance specified in the SLA. However, this method is only reliable if the performance model f_i does not alter between the stand-alone VNF topology and the chained VNF topology. It is possible that the stand-alone performance trends in Fig. 6a deviate, once the VNFs are placed into a chain. There are situations where there is large mutual influence between VNFs in the chain (e.g. VNFs deployed on the same server may show resource contention). In such cases, we cannot any more consider the VNFs to be isolated from each other.

A possible solution is to consider the VNF chain as a closed system, a black-box with the resulting performance of the combined VNF_i's. Then we need to predict a performance model f_{chain} for the total VNF chain, in order to do a correct estimation of resource allocations. In the next section, we will investigate if f_i can indeed vary when VNF_i is placed into a chain.

5. From Stand-alone to Chained VNFs

To create a large pool of test data and possible VNF combinations, we test four different VNF chains. It is the intention to check if the performance profile f_i changes significantly, once the VNF is placed into a chain. We profile each VNF, described in Section 3.1, stand-alone. Then we create chains with the VNFs grouped per two:

- Tinyproxy (VNF1) + Haproxy (VNF2), and vice-versa. These are Docker container based VNF chains.
- pfSense (VNF1) + OvS (VNF2), and vice-versa. These are VM based VNF chains.

5.1. Test Automation

Every stand-alone or chained topology is tested using the same parameter range. The following table summarizes the settings to create our datasets. The values in Table 2 are chosen to reflect an operational space as large as possible, within the limits of the available test hardware:

parameters in f_i or f_{chain}			**values**
input	workload (wl)	*conc. reqs.*	$[1-1000]$ (30 values, evenly along the log scale)
		filesize (MB)	$[0.5, 1, 5, 10]$
	resource (res)	$vCPU_i$ (%)	Haproxy = $[25, 50, 75, 100, 200]$ Tinyproxy = $[25, 50, 75, 100, 200, 300, 400, 600]$ pfSense = $[25, 50, 75, 100]$ OvS = $[25, 50, 75, 100, 200]$
output	performance ($perf$)	95% *resp. time* (ms)	averaged over 5 repetitions of the input space

Table 2: The parameter space for f_i and f_{chain} models.

When considering all of the above mentioned configuration options (stand-alone and chained), we arrive at a total of 14400 sample points. Moreover, each possible resource and workload combination is profiled five times to filter measurement noise as good as possible, without extending the profiling time too long. We build our models using the averaged performance metric (response time) over these five repetitions. Each sample point can take up to $60s$ to ramp up and stabilize, yielding a total test time of several weeks. We can gain some time by distributing the tests over multiple identical compute nodes using our profiling tool described in [25].

The measured KPI is the 95th percentile of the response time, meaning that 95% of the requests have a response time smaller than the reported value. Other publications [9] [11] use the mean response time as KPI. From the viewpoint of a service provider, the mean value is not as interesting, as outliers may still deteriorate the overall service quality. Our measurements also show that the distribution of the response time can be quite skewed and long-tailed, hence the 95% response time provides a more realistic KPI instead of the mean value. Since enough time is taken (up to $60s$) to let each sampled input configuration stabilize, the monitored value of the 95% response time is taken over a large pool of completed requests during this period, and we can assume that single outliers are filtered.

For each unique combination of *vCPU* allocation and *filesize*, we vary the number of *concurrent requests* as specified. In order to automate the measurements, we build in some mechanisms avoid that too much time is spent measuring unuseful situations: It can happen that a configuration could be not profiled up to 1000 concurrent requests: We configure the automated test in such a way that a timeout occurs after $60s$. If then the KPI metrics have not stabilized, or the client/server containers are saturated, then the measurement is not recorded and we move on the next workload to test. Also response times $>$ 20s are not included in our data, as these are considered beyond useful service performance. This filters about 20% of the sample points.

Figure 7 illustrates several subsets of each VNF chain the measured dataset.

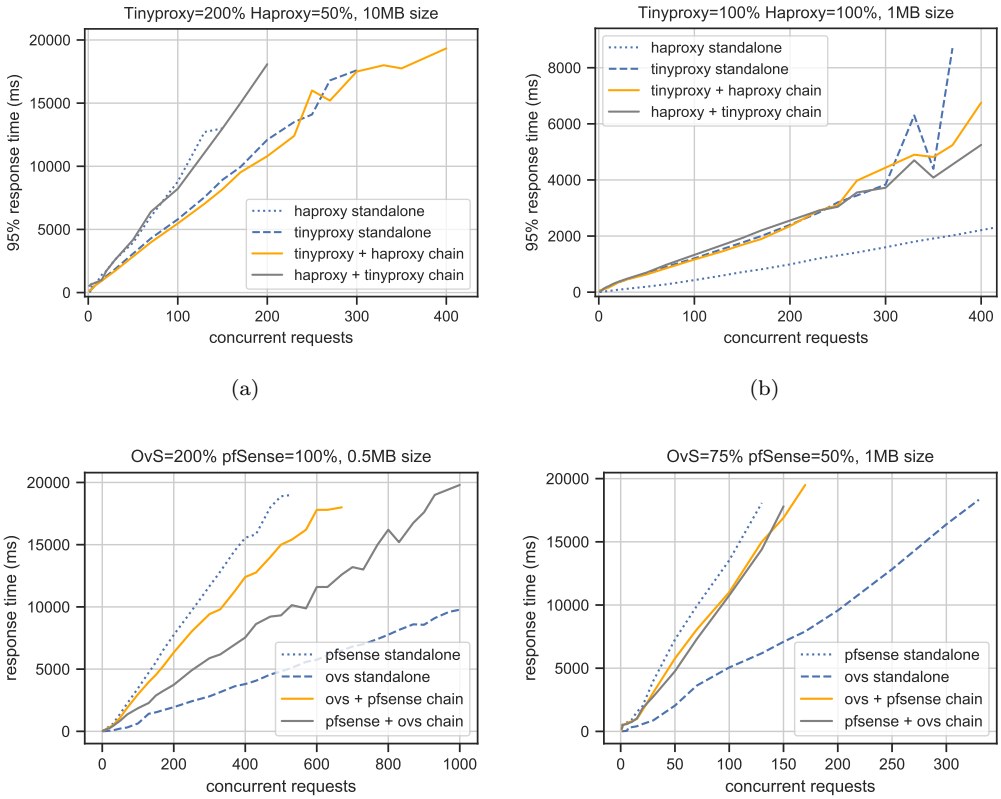


Figure 7: Example performance trends of stand-alone and chained VNFs. The vCPU allocation and requested filesize is in the plot title. The maximum of both dashed curves is the expected chain response time. In (a), (c) and (d) the chain performance deviates from the expected value.

5.2. Model Validation

We first check the modelling accuracy of the methods proposed earlier in Section 3.3. In Table 3, we first test the accuracy of the model f (Eq. 1), trained with the

available data from chained and stand-alone VNF topologies. The reported score values are averaged over the whole range of profiled input parameters (listed in Table 2). A five-fold cross validation is done to asses the accuracy. This means that the dataset is split into five equal parts. Each part (20% of the data) is once selected as test set, while the other 80% of the data was used as training set. The RMSE is averaged over those five validation runs. The result is a baseline RMSE accuracy which can be used for two things:

1. An indicator to compare which method would work best to model the typical trends in our datasets. The bottom row in Table 3 is the average of the upper scores in each column. This allows us to compare the different methods to each other.

2. A baseline benchmark value for the RMSE, to compare with later, when we predict f_{chain} , using only data from f_i.

tested topology	generic method RMSE			linear regression RMSE		
	Interpo-lation	Gaussian Process	ANN	Lasso3	Elastic Net3	Multi Lasso2
Haproxy-Tinyproxy	606	1129	1579	1447	3170	509
Tinyproxy-Haproxy	582	2260	1243	1293	3041	504
pfSense-OvS	993	2765	2678	1574	4505	1443
OvS-pfSense	868	2736	2376	1153	4413	1091
Haproxy	417	1743	1154	904	2859	367
Tinyproxy	690	939	1642	1195	3261	671
pfSense	1119	2554	2524	891	4082	808
OvS	525	1694	2089	1305	4026	644
average(ms)	**725**	1978	1911	1220	3670	755

Table 3: Averaged RMSE of the direct model f (Eq. 1) when trained by different methods (prediction error on the 95% response time (ms))

In Table 3 we compare on the left more generic methods, capable of fitting non-polynomial trends. When we repeat the calculation multiple times we see that the RMSE varies greatly for different tries of the ANN method, indicating that the training procedure does not converge. ANN is the worst method here because we lack sufficient samples to sufficiently train this model.

On the right side, we compare linear regression based methods, capable of modelling polynomial trends. Elastic Net3 is performing worst here, probably because it has the least feature selection capability to filter only the most correlated terms from the third order polynomial expansion. Lasso can regulate the coefficients more aggressively.

We see that linear regression based methods are providing better fits. We can conclude from Table 3 that the Interpolation and Multi Lasso2 method are providing the best fit to the actual measured data of the stand-alone and chained topologies. These methods are based on the ones which performed best in earlier research [9]. The averaged and thus noise-filtered samples from the profiling seem to represent the linear character of the trends witnessed in Fig. 7. This makes the *Interpolation* method a good candidate to

provide a baseline truth model as benchmark. The cross-validation method also confirms this, as *Interpolation* shows the best RMSE score.

5.3. Chained VNF Performance Prediction

We make a distinction between the performance model for stand-alone VNFs (f_i), and the performance model of a complete VNF chain (f_{chain}). Our goal is to find a method F which is able to predict f_{chain} from its stand-alone VNF models f_i. Note that a stand-alone model f_i should be available for each VNF_i in the tested chain. For the total chain we now validate following model:

$$f_{chain}(wl, res) = F(f_1, f_2, ...) = max(f_1, f_2, ...) = perf \tag{4}$$

As a starting point, we assume the most logical assumption, that the total chain performance is limited by the weakest link in the VNF chain. If we assume that each VNF_i operates independently from the other ones, with no mutual influence, then the response time of the whole chain matches the worst-case response time of the stand-alone VNFs. The response time of the total chain can then be estimated in Fig. 7 as the maximum value of both stand-alone VNF_i trend lines (the two dashed lines). At each input value, the function F in Eq. 4 selects the maximum response time of f_i of VNF_i in the chain.

The validity of F in Eq. 4 as 'the weakest link function', is first visually checked on some random profiled chain configurations in Fig. 7. Moreover, we notice several deviations for the predicted chain performance: the weakest stand-alone VNF performance is not always matching the performance of the VNF chain. In some cases the chained performance is even better, outperforming the stand-alone profiling test with lower response times.

As was mentioned in section 4.3, we have now proof that stand-alone VNF models f_i change, when the VNFs are placed in a chain. These phenomena seem to confirm earlier experiments done in [10] and [11], and we analyse this further in Section 6. The example of Fig. 7a will be inspected more in detail in Section 6.2. But first, we want to get a better idea how often these large errors occur. Since it becomes infeasible to visually inspect all profiled configurations, we must revert to other methods to analyse the overall accuracy. In the next subsection we check how frequent and large the prediction errors of f_{chain} are, over all tested chains and configurations.

We now check further how accurate the prediction by F is (in Eq. 4), when every f_i is modelled by the *Interpolation* method. Table 4 compares the prediction of f_{chain} to the baseline RMSE from previous table (we added the last three columns in Table 4 for comparison). We see that the RMSE increases significantly when f_{chain} is used as prediction for the chained performance.

chain	f_{chain} prediction			directly trained		
	Interpolation			chain	VNF1	VNF2
	MAE	MAD	RMSE	RMSE	RMSE	RMSE
haproxy-tinyproxy	577	87	1242	606	417	690
tinyproxy-haproxy	630	168	1354	582	690	417
pfSense-OvS	1458	648	2417	993	1119	525
OvS-pfSense	1071	690	1924	868	525	1119

Table 4: Absolute errors of f_{chain} (Eq. 4) when modelled by the *Interpolation* method (prediction error on the 95% response time (ms))

The results in Table 4 suggest that the average absolute error is increasing substantially, when the 'weakest link prediction' in f_{chain} is used. This is however an averaged value over all tested workloads and resource allocations and it would be interesting to know if this large prediction error holds for all profiled configurations, or only a few outliers. Further tests are motivated by following assumption: In Table 4, the large deltas between the mean (MAE) and median (MAD) indicate a right skewed, longer tailed distribution of the prediction errors. Moreover, large differences between the RMSE and MAE often indicate that outliers are present in the residuals. This observation triggers further investigation to find out where the largest errors are occurring.

A possible application of the model f_{chain} is as part a resource recommendation as in Eq. 2. We discussed in Section 3.2 that this optimization problem could be solved by iterating through all profiled resource and workload configurations. Therefore it makes sense to check if the accuracy of f_{chain} is fluctuating a lot over the different configurations of $filesize$ and $vCPU_i$. Additionally we also switch to the relative error (MAPE). This provides a clearer interpretation how far the prediction is off. Therefore we plot in Fig. 8 the MAPE score. On the x axis, each point represents one possible configuration, i.e. a unique combination of requested *filesize* and $vCPU_i$ allocation for each VNF in the chain. As suspected from the selected measurements in Fig. 7, we also see large prediction error variations in Fig. 8. The MAPE is calculated using again a five-fold cross validation of the profiled data. The given *baseline chain* and *stand-alone* accuracies in Fig. 8 are modelled using the *Interpolation* method, as explained before. The *predicted chain* depicts the accuracy of f_{chain}, taking the max response time of the shown stand-alone models.

At first sight, the predicted chain performance can have large errors, and this seems to be true for all tested chain configurations. In general, our results thereby confirm the research from other authors in [10] and [11]. We will dive deeper in the causes of these prediction errors in Section 6. To anticipate things, we can mention here that it is very hard, if not impossible, to emulate all factors influencing the bandwidth in the chained VNF topology, during stand-alone testing. This leads to deviating performance between tested topologies. It is also clear from Fig. 8 that the prediction errors of F are not caused by any bad underlying models, as the stand-alone models f_i show way better accuracy. The depicted baseline chain performance is an *Interpolation* model trained by the actual measurements of the chain's response time. This is thus a baseline or best-case scenario, where training samples are very representative. Still in Fig. 8, the *predicted chain* performance (f_{chain} by F) is under-performing. Which indicates that

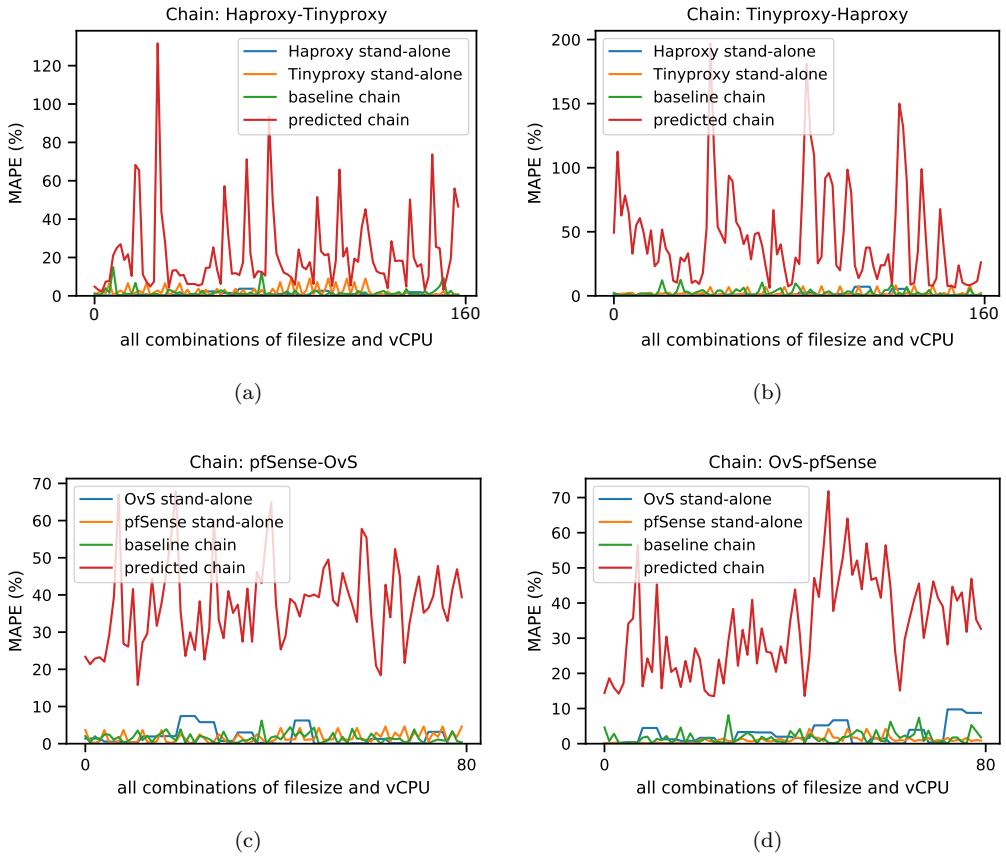


Figure 8: The predicted f_{chain} performance by F (Eq.4) has room for improvement compared to the baseline chain model and is significantly worse than the underlying stand-alone VNF models f_i.

the training data here, being the max response time of the stand-alone VNF models, is not guaranteed to be a good predictor. But we know that it can be improved up to the baseline performance by taking new training samples. To improve the accuracy, and still profit from the stand-alone test data, we look for a method to adjust our initial chain prediction with newly measured samples. In the Section 7 we will therefore analyze ways to online adjust the predicted model F to reach a better fit for f_{chain}. But first we analyze in Section 6 why F is not a good predictor for f_{chain}.

6. VNF Performance Variation Analysis

In an attempt to understand the root causes of the prediction errors by F in the previous section, we dive a bit deeper into the data path of chained VNFs. We suspect that several optimizations and offload mechanisms implemented in the operation system's kernel, contribute to the unpredictability of VNF chain performance. Ths is the root

cause why f_i tends to vary when VNF_i is placed into a chained topology. We provide some technical background here. There are several stages in the networking stack of the Linux kernel as given in Fig. 9. The common processing path traverses following steps [37]:

1. The VNF application creates socket buffers to read/write data to.
2. The kernel will then periodically write/read data to/from these socket buffers, taking care of the further data transmission and reception between the chained VNFs.
3. This data is then sent to the next VNF in the chain, or recieved from the previous one, over the TCP/IP protocol implemented in the network layer of the kernel.
4. The actual transmission/reception over the network's physical layer is done by the hardware of the network card.

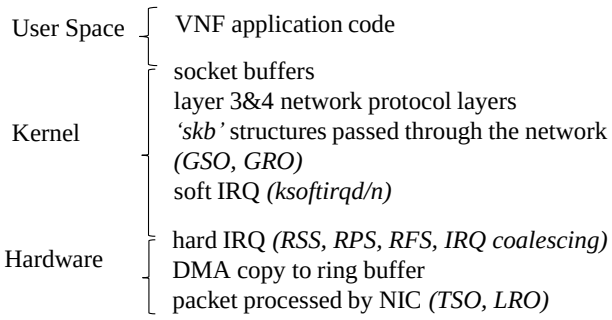


Figure 9: Different processing stages of packets through the Linux kernel network stack.

6.1. Packet Coalescing Effects

The processing overhead in the kernel is mainly packet based (functions are called per packet, irregardless of the packet size). To mitigate this, a whole range of optimization mechanisms is implemented, which aim to coalesce packets as much as possible, creating larger packet sizes and thus less packet-based overhead (but also creating a trade off regarding information corruption in case of packet reordering or loss). But the main advantage is that less processing from the CPU is required, leading to an overall increase in throughput. Note that this is not limited to virtual interfaces alone: also if the VNFs are located on physically separate servers, the same offload mechanisms take place for traffic over physical interfaces. Of course, the configured Ethernet Maximum Transfer Unit (MTU) should be respected, and packet segmentation is eventually offloaded to the (virtual) network interface. Without diving into the exact implementation details, we can briefly mention the following mechanisms which aim to optimize the standard TCP/IP processing:

- On layer 4, the kernel tries to estimate the bandwidth-delay product of each TCP connection, in order to optimize receive and send windows [38] [39] [40]. This is

done to maximize the amount of data being sent over the link, by minimizing the time the TCP connection is idle, waiting for acknowledgements. Due to these larger TCP windows, larger chunks of data are being sent.

- The larger chunks of data, originated as explained in previous point, are not immediately segmented into smaller packets. The kernel keeps the network packets as large as possible by combining them, up to certain size and timeout thresholds. Final packet segmentation happens as late as possible in the network stack, preferably offloaded to the (virtual) NIC. TCP Segmentation Offload (TSO) is the NIC driver implementation and Generic Segmentation Offload (GSO) is the kernel implementation of this mechanism.

- Similar to the previous point, Large Receive Offload (LRO) and Generic Receive Offload (GRO) implement packet coalescing on the receiving path [41]. Packets are combined as early as possible as they enter the network stack. These methods reduce the number of packets passed up the network stack, by combining "similar enough" packets (in terms of header fields and timestamps), and thus reducing CPU usage.

The command *ethtool* can be used to modify the offload settings for the VNF interfaces. Note that large packet sizes (> MTU), are most likely to be monitored because the system has GRO/GSO enabled (by default). This happens because packet capture taps are inserted further up the stack, after GRO/GSO has already happened. The effect of the packet offloading can be seen in Fig. 10: As an example we show the averaged packet sizes monitored in the Haproxy VNF, in the different tested topologies. There is a clear difference in packet size between the stand-alone and chained configurations. At lower workloads on the x axis, large TCP windows enable GSO/GRO to create larger packets. But at higher workloads, more concurrent TCP connections need to be maintained by the VNFs, overloading the allocated vCPU (opening/closing sockets, transferring data in user space between sockets, user/kernel context switches etc.). As each TCP connection receives less CPU time, TCP congestion control dynamically lowers the windows, and thus smaller chunks of data are being sent. The packet sizes coalesced by GSO/GRO are thus more limited at high VNF loads. In Fig. 10 we see the largest impact on the ingress traffic, which is a result from a smaller TCP receive window, advertised by the receiving VNF. When sniffing the traffic, we indeed witness occasional *TCP Zero Window* messages. This means that a client is not able to receive further information at the moment, and the TCP transmission is halted until it can process the information in its receive buffer.

We have also tested Layer 2 VNF chains (pfSense + OvS), In which TCP endpoints only exist in the clients and server. These setups show large errors as well, when considering the stand-alone tests only to predict the performance of the chain. This can be explained because the bandwidth-delay product of the TCP connections increases, while passing through highly loaded layer 2 VNFs. Thus the TCP windows in the sending and receiving application (server, clients) are adapted dynamically, causing GRO/GSO to lower packet sizes in the TCP endpoints. The chained topologies thus create a different bandwidth-delay product as compared to the stand-alone topologies, causing different profiling results between the two environments.

Packet sizes also seem influenced if the VNF order is changed or if more VNFs are placed in the chain. This can be explained because each of the VNF's ingress and egress interfaces has their own GRO/GSO threads to modify packet sizes differently, for the same bandwidth. For example, the same workload and bandwidth in a stand-alone and chained topology, can be transferred by different packet sizes and packet rates depending on the VNF order or number of VNFs in the processing chain. The whole VNF chain (including client and server applications) is thus a balanced system where the processing delay and dynamic buffers such as window sizes of one VNF determine the processing of the next VNF in the chain. This interdependency makes it very hard to predict overall chain performance.

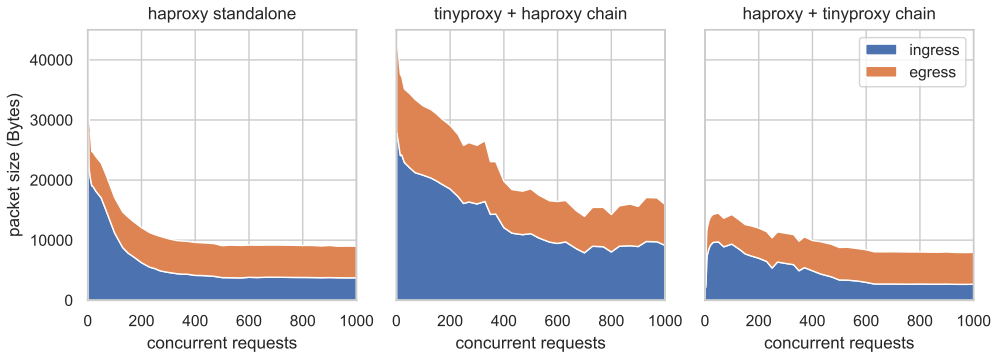


Figure 10: Variation of the Haproxy VNF ingress/egress packet size, in several profiled topologies.

Note that for multi-queue enabled NICs there are also other optimization mechanisms available in the Linux kernel [42]. This is however not enabled in our experiments. In short, we can mention that techniques such as:

- *Receive Packet Steering (RPS)* : A hash function is calculated over the header fields of each incoming packet, for example, a 4-tuple hash over IP addresses and TCP ports of a packet. The goal is that packets belonging to the same flow are hashed the same and processed by the same CPU. Spreading load between CPUs decreases the queue length.

- *Receive Side Scaling (RSS)*: The hardware implementation of the hashing function in the NIC, so it can be offloaded from the CPU.

- *Receive Flow Steering (RFS)*: A more advanced form of RPS, but next to hashing also "application locality" is taken into account. The goal of RFS is to increase datacache hitrate by steering kernel processing of packets to the CPU where the application thread consuming the packet is running. RFS relies on the same RPS mechanisms to enqueue packets onto the backlog of another CPU and to wake up that CPU. In RFS, packets are not forwarded directly by the value of their hash, but the hash is used as index into a flow lookup table. This table maps flows to the CPUs where those flows are being processed.

The above mechanims complement the Linux networking stack to increase parallelism and improve performance for multi-processor systems. There are some caveats here regarding VNF profiling: CPU allocation should again be carefully isolated, so traffic forwarding in the kernel can be differentiated from VNF internal processing.

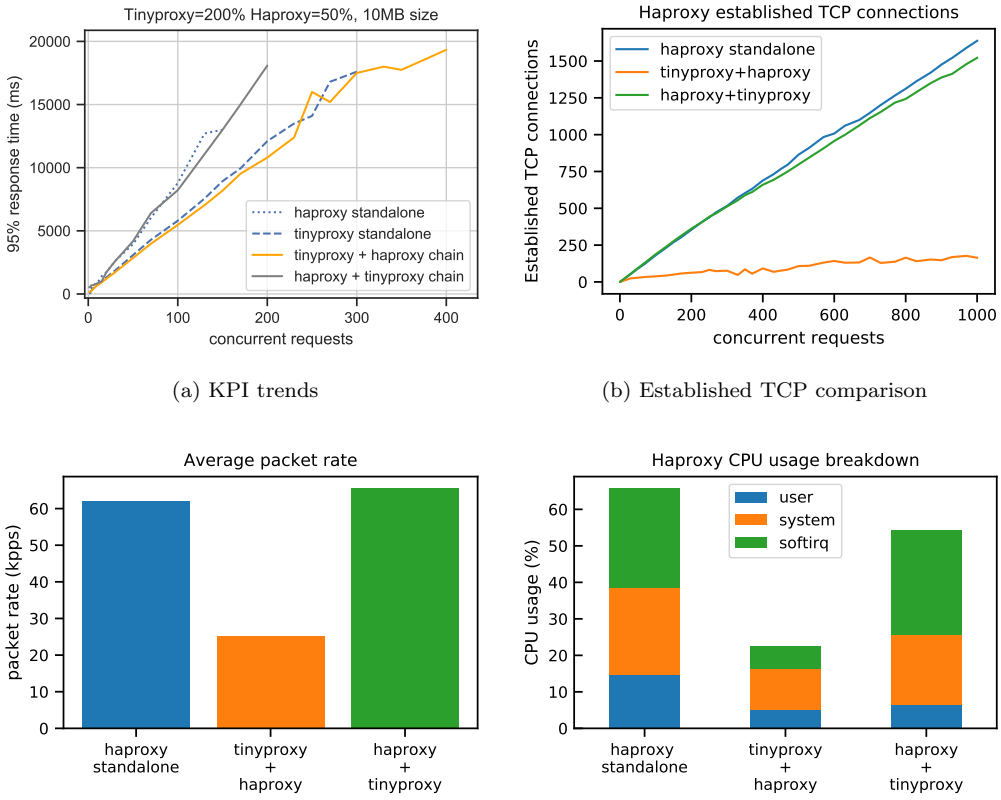
6.2. Stand-alone vs. Chained Test Context

We started our paper by visually inspecting the sample measurements in Fig. 7: performance trends seemed to vary whether tested in the context of either stand-alone or chained topologies. Since the workload and resource allocation is the same, other factors seem to influence the chained performance, which were not profiled in the stand-alone tests. It proves to be very hard to mimic the same operational context of a chained topology in a stand-alone setting, due to many influencing factors. We retake the example measurement of Fig. 7a and in Fig. 11 we illustrate the varying environment parameters more closely. Our initial observation is Fig 11a, where we see that the performance as measured in a stand-alone topology, can improve if the VNFs are placed in a different order. In Fig. 11 b,c,d we look at how several parameters of the Haproxy VNF in these chain configurations differ between tested topologies. The chain topology with the best performance (Tinyproxy+Haproxy) shows the least overhead in the figures. We distinguish below list of indicators which show that the same workload and resource allocations are loading the VNF in a different way, depending of the profiled topology:

The number of concurrent established TCP connections. The creation and closing of TCP connections is mainly controlled from the VNF application itself. For example, the VNF can choose to limit the number of open TCP connections to have a more efficient bandwidth usage [43]. In Fig. 11b we can see that the number of established TCP connections differs, depending on the chain configuration (exemplified by the averaged number of established TCP connections for Haproxy in different profiled topologies). This indicates that the VNF operation is clearly different depending on the chain configuration.

Network packet rates. As explained in previous subsection, offloading mechanisms (GSO, GRO) dynamically change the packet sizes. In Fig. 10 we already saw that the packet size reaches different equilibria depending on the chain topology . By sending or receiving the same file requests using larger packets, the packet *rate* decreases, resulting in a lower processing load of the ingress/egress network packets. If we look at the averaged combined packet rate per topology in Fig. 11c, we also see clear differences per tested topology. This indicates that for the same workload, different packet sizes are being used.

VNF CPU usage breakdown. Sending and receiving packets through the network stack generates a special type of interrupts in the Linux kernel: *software interrupts (softirqs).* These interrupts point to certain queued tasks in the kernel, of which packet transmit and receive to/from (virtual) network interfaces is of interest here. Each CPU has its own *ksoftirqd/n* kernel thread (where n is the logical number of the CPU). Each *ksoftirqd/n* thread gets a share of pending softirqs scheduled and executes their functions [37]. These kernel threads can be monitored and are reported as softirq cpu usage, the relative cpu time spent executing softirqs. For networking intensive applications, high softirq cpu usage is thus an indication for high networking overhead [44]. The relative differences in packet rate in Fig. 11c are reflected in the CPU usage breakdown in Fig. 11d. Although



(a) KPI trends

(b) Established TCP comparison

(c) Packet rate comparison (ingress+egress)

(d) CPU usage comparison

Figure 11: Comparison of Haproxy VNF performance, in different profiled topologies.

Haproxy was allocated a share of 50% CPU, we monitor a higher percentage usage of this CPU in some cases. Deviations here can be explained due to Linux's implementation of softirq handling: resource isolation between VNFs (containers) on a single server is not perfect when it comes to networking . It seems the kernel is always able to schedule the handling of softirqs to any CPU, even when this CPU is allocated to other processes [45]. But again we can conclude from Fig. 11d that the VNF is differently loaded, influenced by the chained topology.

In the above analysis, deeper inspection showed that a VNF's processing load can differ greatly for the same workload and resource allocation, but in a different chain topology. One could consider to incorporate the above indicators in the profiled model, to improve chain performance predictions. Profiling all of these factors is however an unrealistic endeavour since they are not independent and are caused by hard to control parameters such as delay, bandwidth availability, buffer sizes or specialized OS or VNF configurations. The possible configuration space would likely become infeasibly large to profile. Secondly, we would enlarge the input dimensionality of the performance model

which would require extra training samples and preferably a form of feature selection to improve the model accuracy [46]. For accurate feature selection, the total chain would need to be profiled also, to check which factors are most correlated to the chain KPIs, and rule out any collinear factors. So in total, we would not save any profiling effort by including these extra factors in our model. Additionally VNF performance on different cloud providers can also show large deviations [47]. So for accurate VNF performance predictions across cloud providers, all of them should be profiled in advance. These reasons all further advocate for an online adjustment system for the performance model, due to unpredictable factors only known at time of deployment.

7. Enhancing the Chained VNF Prediction

To avoid an expensive profiling phase of the total chain, we want to keep benefiting from the stand-alone VNF profiles f_i to obtain f_{chain}. But it is now also clear that the accuracy of F is not sufficient without further improvements. Once the total VNF chain is deployed by the service provider we propose that f_{chain} is online re-trained, starting from the initially predicted model by F. Despite f_{chain} not being completely accurate initially, we show that a relatively small amount of extra samples can raise the accuracy to more acceptable levels. This online retraining phase is still more efficient than a complete chain profiling test. Following subsections validate different strategies, to improve the chain model accuracy.

7.1. Align the Training Data

There are also other aspects which can improve the prediction accuracy. The goal of the performance model f_{chain} is to predict where the KPI limit (max response time in the SLA) will be hit. So for good prediction accuracy, best chances are to include enough training samples in the neighbourhood of that targeted limit. By aligning the training data more around the target response time, we have a higher chance that the local performance trend is correctly modelled. Depending on the use-case, only a specific KPI range may be of interest. For example, if we want to model where the response time reaches $1000ms$, workload samples reaching $1ms$ response time are less likely to contribute much to prediction accuracy at higher workloads and can even make this prediction worse.

In Fig. 12 we investigate the accuracy of f_{chain} in different buckets of the response time (Note that the scale on the y-axis is logarithmic). Since we have test data in a large workload span (up to 1000 concurrent requests), we also have a large range of response time values (from $1ms$ up the order of $10000ms$). We can see in the figure that no modelling method can achieve high accuracy in every KPI bucket. Our profiled dataset is giving worse accuracy in the first two bins because:

1. Relatively less samples are available in those lower response time ranges. Our tested workloads produce mainly response times in the higher ranges.

2. As exemplified in [9], a different performance trend is likely predominant at low KPI values (before resource saturation). Since in our dataset, more samples are available at higher response times, only the main trend at higher response time values is accurately captured by the model.

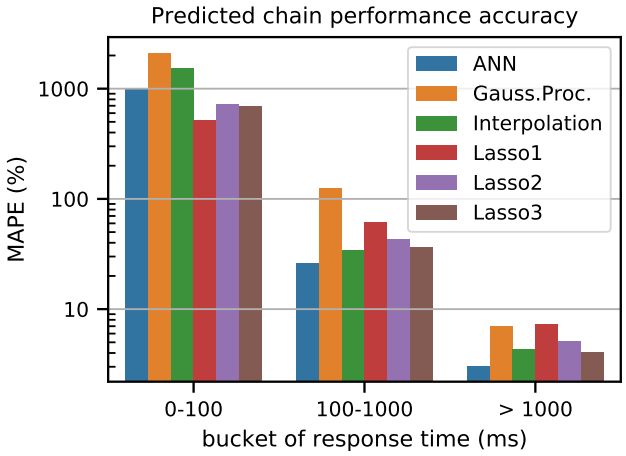


Figure 12: The prediction accuracy is not good at low response times.

Since there are only few samples in the lower buckets, the bad prediction outliers of f_{chain} in Fig. 8 cannot all be attributed to this effect. But nonetheless, it is clear that the best prediction accuracy is obtained if enough training data is available in a range around the KPI target. For our profiled dataset, we choose best a response time target $> 1000ms$, as suggested by Fig. 12. Note that if lower response times need to be modelled, we need to focus our training set more in this area, This can be done by filtering training data only in these bins and steer the profiling phase to generate more workload samples which produce lower response times. Since a choice has to be made for optimal prediction accuracy, for our further validations, we chose not to train our prediction model for such low response times and we omit training data $< 100ms$, as this is well below our wanted max response time target.

7.2. Online VNF Chain Training

When retraining f_{chain} using online production data, we cannot expect that all workload parameters are varying in the same range as generated during offline profiling. We can make some assumptions: The online workload in the production environment will only show variation in a smaller subset of the total space of profiled workloads. Typically, the generated workloads for VNF profiling span a broader range, including the boundary values of the expected workloads in production. This ensures better interpolation accuracy of the prediction model, as also explained in [25]. But during online retraining, not all workload parameters can be controlled like in offline tests. In our experiments for example, we have profiled requests for several filesizes (from 0.5 to 10MByte). But in production the average filesize could remain fairly constant on the long-term. On the other hand, the number of concurrent requests is more probably a short-time varying workload parameter. This means that online retraining of a pre-profiled model will probably only happen in a certain subspace of the model, namely in the parameter range where short-term variations take place in the online workload. Using this assumption, we can filter the stand-alone VNF data and slim down the prediction model by keeping

only the input parameters which are expected to vary on the short-term. In general, by aligning the stand-alone VNF profiling data with the expected workload in production, we can limit the number of input variables and thus simplify the model. This benefits both the accuracy of the prediction model and the retraining speed, as exemplified in the next reported figures. We train models where only the number of *concurrent requests* is a varying input parameter. We do this for each unique configuration of *filesize* and $vCPU_i$, and then plot the averaged accuracy over all configurations. This is similar to averaging all points in Fig. 8 per different modelling method, in order to select which method can provide the best overall accuracy.

The offline, stand-alone VNF tests produced a first estimation for f_{chain} by using taking the maximum response time in each tested workload (Eq. 4). These initial samples are taken to train f_{chain} like the VNF chain is a stand-alone system on its own:

$$f_{chain}(wl, res) = perf$$

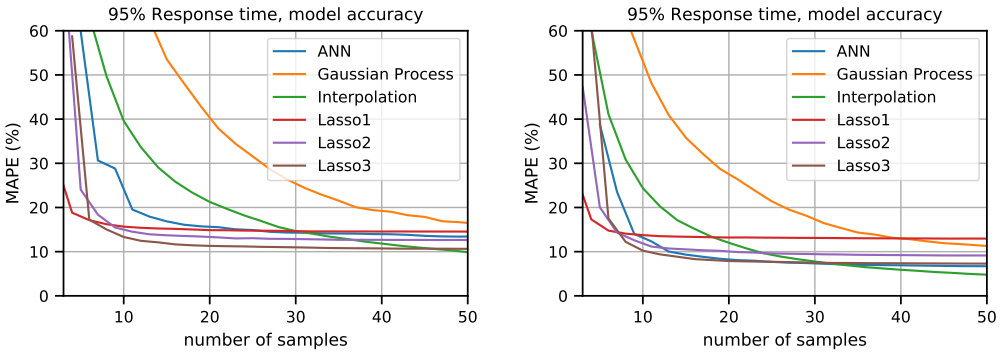
It is the intention that the model f_{chain} is now adjusted in an online way, by taking extra training samples while the service processes production workload. In the previous model validation, the Interpolation method seemed to be the best fit. This is no surprise, as this method is also most prone to overfitting because every training sample is crossed by the regression line. Online modelling methods should have at least some form of regularization, which mitigates measurement noise and overfitting. To rule out large errors in f_{chain} due to overfitting, we should include other methods also in the forthcoming validations where new online training data is used.

We therefore test in Fig.13a which modelling method for f_{chain} reaches the best accuracy while incrementally retrained with more online samples. The workload samples are generated randomly between 0 and 1000 concurrent requests. This is done to simulate a fluctuating workload in production. In Fig.13a we incrementally train the predicted chain model f_{chain}, with samples generated by F in Eq. 4. On the other side in Fig. 13b, we use samples from the actual chain measurement to train the selected models. In each point on the x axis, the accuracy is cross-validated over five different sets of random samples, and averaged over all four tested VNF chains. This to select which method can model best the typical response time trends. The main conclusion from Fig. 13 is that the same models perform equally in both graphs: The model working best on the data of stand-alone profiled VNFs is also providing the best fit for the trend of the measured VNF chain. This indicates that the trend predicted by F from the stand-alone tests and the chain's actual performance trend are modelled best by the same method. This brings extra confidence that the trend *shape* of the chain performance can be estimated from the stand-alone tests, and efficiently retrained afterwards with actual data from the chain.

7.3. VNF Chain Canary Training

In the previous section, random workload samples were used to retrain f_{chain}. This assumes that in production, the workload should vary in a large range, which is not necessarily happening in reality. Also sudden workload spikes can cause unpredicted service performance, if the model was not yet accurately trained in that region.

In order to improve the online retraining procedure in a reliable way, we propose canary testing for a safer retraining phase in production environments [48]. In a canary



(a) Trained on the predicted chain performance

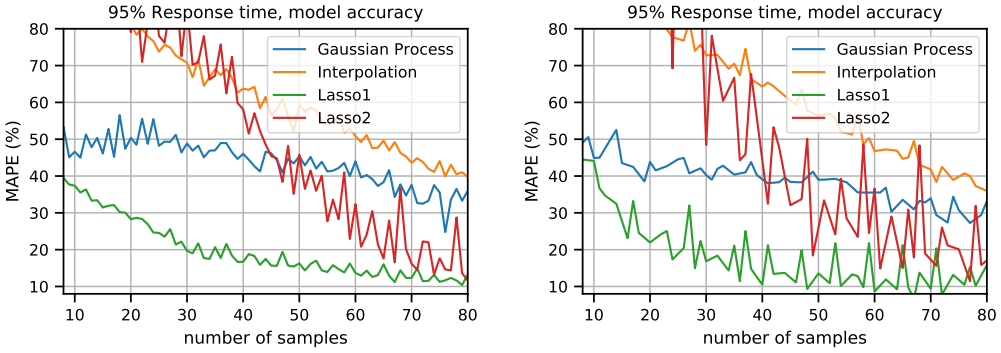
(b) Trained on the chain's actual performance

Figure 13: Comparison of different models, online retrained with random samples.

test, a fraction of the total production workload (e.g. a small number of users) is processed by the service under test. The advantage is that the service is tested using realistic workload and that the possible impact of an underestimated service performance is kept relatively small (e.g. since a smaller number of users are affected). By incrementally allowing more and more workload to the service under test, we can create a controlled test environment similar to the offline profiling. Note that testing lower workloads does not mean that only low response time values are being monitored. (Gathering samples of too low response time was not preferable as explained in previous Section 7.1.) As a compromise we could increase the workload until the measured response time is in the same order of magnitude as the maximum response time in the SLA, and thus samples are taken past the worse bin values as was illustrated in Fig. 12.

When a sufficient range of canary samples is gathered, f_{chain} would ideally predict for higher workloads also. This requires however a model which can reliably extrapolate to higher workloads, beyond the canary training data. This is tested in Fig. 14 on our selected modelling methods. On the x axis, we simply start from the minimal tested workload and gradually increase by extra concurrent requests. We increase samples up to half of the expected workload range (canary goes up to 500 concurrent requests, while the profiled workload can go up to 1000). So the model is trained on the lower half of the workload (the canary workload), while the reported accuracy on the y axis is tested on the upper half of the workload samples, beyond the canary training data. We have left out ANN and Lasso3 models for better readability since those models had the worst accuracy under canary training. The plots are a bit more noisy here compared to previous figure, because we cannot cross-validate over multiple, randomly ordered sample subsets. The order of samples is fixed during canary testing: Here, the samples are taken by ascending workload, incrementing the number of concurrent requests by five in each sample.

The results show which method can predict best the total chain performance, using only the canary workload samples. We see that the Lasso and Gaussian Process methods are providing the best extrapolation accuracy [49], at the lowest amount of canary



(a) Trained on the predicted chain performance

(b) Trained on the chain's actual performance

Figure 14: Retrained models using canary testing with increasing workload samples.

training samples. Other methods need more data before accuracy converges. This is not totally surprising, as these are the regression methods explicitly aiming for a (quasi-) linear trend , which we already assumed from Fig. 7. It can be intuitively imagined that a linear trend is quite easy to extrapolate to untested workloads, because few parameters need to be modelled (only intercept and slope for a one dimensional input). Other (more non-linear) fitting methods are easily overfitting the canary training samples, leading to larger errors when extrapolating. For generic machine learning methods relying on nearby training samples (e.g. nearest neighbours, neural networks or random forests), the extrapolated accuracy converges at least more slowly, or cannot be guaranteed at all. This is because most machine learning regression methods are optimized to fit non-parametric trends, and accuracy depends on the availability of nearby training samples. Thus extrapolation outside the boundaries of the trained workloads is hard in this case. Similar to the previous test, we see that the trend predicted by F from the stand-alone tests (Fig. 14a) and the chain's actual performance trend (Fig. 14b) are modelled best by the same methods. This again indicates that the trend predicted by F, is a good test case to check which model works best for canary training of the actual VNF chain.

8. Discussion and Conclusion

Modern telecom services consist out of multiple chained VNFs. These services could be deployed more efficiently if they had a performance model available. Moreover, a great deal of profiling effort can be saved if the total chain performance were derived from profiling tests done on the stand-alone VNFs. We have experimentally shown that the major benefit of this approach is a faster service deployment, needing less scaling iterations to reach the required performance, specified in the SLA. However, other experiments on chained combinations of four different VNFs (Tinyproxy, Haproxy, pfSense and Open vSwitch) indicate that care must be taken when predicting VNF chain performance, as accuracy is not guaranteed.

8.1. Discussion

In this section we discuss some learnings and rules of thumb we encountered during our validations. In general, we noticed that machine learning-based methods need more training samples. They are converging slower than linear regression-based methods. This is because the regression methods need inherently less training samples to fit the collection of quasi linear trends we witness in our tests. There are also some other learnings we can share:

8.1.1. Sensitivity of the Validated Modeling Methods

Regression methods based on interpolation are notorious for their risk of overfitting the sampled trend. When using profiled VNF data this risk is under control: In the offline profiling environment (Phase 1), we maintain in control of the generated workload. For example under a fixed workload, our profiling setup implements a ramp-up time until performance stabilizes, after which the KPI is measured as an average over a moving time window. This ensures that each taken KPI sample is a good representative value for the given input parameters of workload and resource allocation. Additionally, the measurements can be repeated multiple times for extra confidence and noise filtering. That is why we trust that interpolation based methods create a confident model from offline profiled data.

On the other hand, during online monitoring in production, workload is probably less stable. KPI metrics cannot always stabilize as in the profiling environment. This means that online gathered training data, is likely more noisy. Therefore, the used modelling method should have some form of regularization built-in. In an offline profiling environment on the other hand, regularization is built-in due to the way training samples are being taken, as described above. In our opinion, interpolation methods are less advisable for online sampled training data, since they tend to overfit the measurement noise. We saw in our validations that Lasso-based methods are a better alternative.

8.1.2. Multi-variate Performance Metrics

In this paper we only validated uni-variate performance trends for f_i or f_{chain}. Only the 95% reponse time was considered as performance metric. It is however likely that multiple performance metrics need to be considered for a VNF chain. In this case, the performance metrics are represented as a vector, and multi-variate modelling approaches can be used. Example techniques include Multiple Linear Regression, Principal Component Regression or Partial Least Squares to solve multi-variate linear regression problems or neural networks for non-linear problems with a vectorized output. A discussion and comparison of these methods is beyond the scope of this article, but we can share our thoughts on possible caveats when modelling multi-variate output parameters.

In our tests, using one performance output metric, it became already clear that neural nets are not the most optimal method to use. This suggests that also for vectorized outputs, accuracy would probably decrease, since we would use the same training data to learn the relation with even more output variables. Thus, more weights in the neural net to train, with the same profiled workload data as input. During the profiled workloads, we can always monitor a vector of performance metrics. But when training data is limited, it makes more sense to keep the model as simple as possible and to create separate models, for each distinct performance metric. Intuitively, this would yield more accurate models

which are easier to train, compared to one large vectorized model for all performance metrics together.

8.1.3. Less Efficient Modelling Techniques

One could think of other ways to predict a resource allocation from the profiled data. Opposite to Eq. 1, a generic machine learning approach could be used to recombine the metrics, and directly predict res:

$$\hat{f}(wl, perf) = res$$

We chose not to train a model like $\hat{f}$ because:

1. The parameters in res are often not continuous but categorical in nature. They represent a discrete value of a resource allocation such as the number of CPU cores, or pre-defined resource flavours offered by the infrastructure provider. As a consequence, a range of continuous wl and $perf$ values can map to a single (integer coded) value of res. For a generic, regression based, machine learning method, this would mean that the training samples are following a step-like shape. We know that regression based methods are not good at modelling such discontinuities, certainly when we strive to limit the number of training samples, as in our use case.

2. When considering classifier based methods, we could successfully train a model to predict discrete classes of res_i, but the predictable classes are limited to the ones included in the training set. Trained classifiers can therefore not extrapolate to unprofiled resource allocations, which limits their ability to predict beyond pre-profiled resource allocations.

8.2. Conclusion

We have investigated how the performance of VNF chains can be efficiently modelled, starting from the stand-alone VNF performance models. To mitigate possible prediction errors, we propose an online adjustment procedure, summarized in Fig. 15, retaken from the introduction for clarity and convenience. The profiled data from the stand-alone VNFs is used to model an initially estimated performance trend of the chained service, using the Interpolation modelling method. This estimated performance trend can be used as test case, to check which modelling method would fit best. When the actual chain is deployed, the selected model is transferred and further re-trained with online gathered data. Polynomial expansion, together with the Lasso regression method, provided the best modelling accuracy in our experiments. We have also demonstrated canary testing to gradually retrain the performance model of the VNF chain. Using this approach, we only need a small range of training samples to extrapolate the performance trend to untested higher workloads. Other analysed methods from the machine learning field are not able to extrapolate well in this use case.

We have experimentally shown that the major benefit of this approach is a faster service deployment, needing less scaling iterations to reach the required performance, specified in the SLA. However, other experiments on chained combinations of four different VNFs (Tinyproxy, Haproxy, pfSense and Open vSwitch) indicate that care must be taken when predicting VNF chain performance, as accuracy is not guaranteed. To mitigate possible prediction errors, we propose an online adjustment procedure, summarized

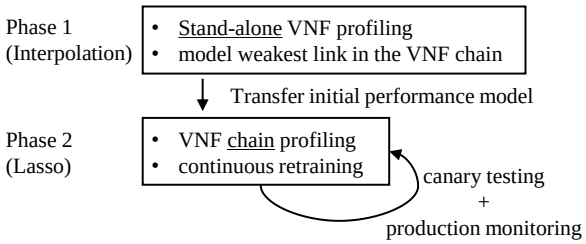


Figure 15: Overview of the presented profiling workflow.

in Fig. 15. The profiled data from the stand-alone VNFs is used to model an initially estimated performance trend of the chained service. This estimated performance trend can be used as test case, to check which modelling method would fit best. When the actual chain is deployed, the selected model is transferred and further re-trained with online gathered data. Polynomial expansion, together with the Lasso regression method, provided the best modelling accuracy in our experiments. We have also demonstrated canary testing to gradually retrain the performance model of the VNF chain. Using this approach, we only need a small range of training samples to extrapolate the performance trend to untested higher workloads. Other analysed methods from the machine learning field are not able to extrapolate well in this use case.

Further applications of our presented profiling workflow are situated around the development and deployment of VNF chains. To optimize the time needed to validate whole VNF chains, performance requirements for stand-alone VNFs can be delegated to multiple developing parties, using a DevOps approach. When the service provider later combines multiple VNFs into a chain, the stand-alone VNF profiles can be used to estimate an initial resource allocation (in function of the expected workload and specified SLA). Finally, during operation, the SLA of the chain can be further assured by online retraining of the chained performance model, derived from the stand-alone VNFs. This additional online retraining mechanism ensures that the performance estimation of the VNF chain can be corrected in production.

Acknowledgements

This work has been performed in the framework of the NGPaaS and 5GTANGO project, funded by the European Commission under the Horizon 2020 and 5G-PPP Phase2 programmes, resp. under Grant Agreement No. 761 557 and 761 493 (http://ngpaas.eu) (https://www.5gtango.eu). This work is partly funded by UGent BOF/GOA project 'Autonomic Networked Multimedia Systems'.

References

[1] S. Van Rossem, B. Sayadi, et al., A vision for the next generation platform-as-a-service, in: 5G World Forum (5GWF), IEEE, 2018. doi:10.1109/5GWF.2018.8516972.

[2] A. Wiggins, The twelve-factor app–a methodology for building software-as-a-service apps (2017). URL https://12factor.net/

[3] 5G-PPP Software Network Working Group, Cloud-native and verticals' services (5gppp-software-network-wg-white-paper-2019_final.pdf), Tech. rep., 5G-PPP Software Network Working Group, [Online; accessed 2020-02-05] (2019).
URL https://5g-ppp.eu/

[4] S. Van Rossem, et al., Introducing development features for virtualized network services, IEEE Communications Magazinedoi:10.1109/MCOM.2018.1600104.

[5] M. Peuster, S. Schneider, M. Zhao, G. Xilouris, P. Trakadas, F. Vicens, W. Tavernier, T. Soenen, R. Vilalta, G. Andreou, et al., Introducing automated verification and validation for virtualized network functions and services, IEEE Communications Magazine 57 (5) (2019) 96–102. doi:10.1109/MCOM.2019.1800873.

[6] M. K. Weldon, The future X network: a Bell Labs perspective - Chapter 13: The future of network operations, CRC press, 2016.

[7] B. Berde, S. Van Rossem, A. Ramos, M. Orrù, A. Shatnawi, Dev-for-operations and multi-sided platform for next generation platform as a service, in: 2018 European Conference on Networks and Communications (EuCNC), IEEE, 2018, pp. 1–5. doi:10.1109/EuCNC.2018.8443272.

[8] J. Grohmann, P. K. Nicholson, J. O. Iglesias, S. Kounev, D. Lugones, Monitorless: Predicting performance degradation in cloud applications with machine learning, in: Proceedings of the 20th International Middleware Conference, 2019, pp. 149–162. doi:https://doi.org/10.1145/3361525.3361543.

[9] S. Van Rossem, W. Tavernier, D. Colle, M. Pickavet, P. Demeester, Profile-based resource allocation for virtualized network functions, IEEE Transactions on Network and Service Management (2019) 1–1doi:10.1109/TNSM.2019.2943779.

[10] J. Nam, J. Seo, S. Shin, Probius: Automated approach for vnf and service chain analysis in software-defined nfv, in: Proceedings of the Symposium on SDN Research, ACM, 2018, p. 14.

[11] M. Peuster, H. Karl, Profile your chains, not functions: Automated network service profiling in devops environments, in: Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE, 2017. doi:10.1109/NFV-SDN.2017.8169826.

[12] P. Xiong, C. Pu, et al., vperfguard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments, in: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, ACM, 2013, pp. 271–282.

[13] J. O. Iglesias, et al., Orca: an orchestration automata for configuring vnfs, in: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, ACM, 2017, pp. 81–94.

[14] A. Abdelsalam, P. L. Ventre, C. Scarpitta, A. Mayer, S. Salsano, P. Camarillo, F. Clad, C. Filsfils, Srperf: a performance evaluation framework for ipv6 segment routing, arXiv preprint arXiv:2001.06182.

[15] The 5g infrastructure public private partnership, https://5g-ppp.eu/, accessed: 2020-02-05.

[16] F. Luque-Schempp, P. Merino-Gómez, L. Panizo, et al., How formal methods can contribute to 5g networks, in: From Software Engineering to Formal Methods and Tools, and Back, Springer, 2019, pp. 548–571.

[17] Y. Jia, C. Wu, Z. Li, F. Le, A. Liu, Online scaling of nfv service chains across geo-distributed datacenters, IEEE/ACM Transactions on Networking 26 (2) (2018) 699–710.

[18] A. Xie, H. Huang, X. Wang, Z. Qian, S. Lu, Online vnf chain deployment on resource-limited edges by exploiting peer edge devices, Computer Networks (2019) 107069doi:https://doi.org/10.1016/j.comnet.2019.107069.

[19] Y. Yue, B. Cheng, X. Liu, M. Wang, B. Li, Dynamic vnf placement for mapping service function chain requests in nfv-enabled networks, in: Companion Proceedings of the Web Conference 2020, 2020, pp. 44–45.

[20] R. Cziva, D. P. Pezaros, Container network functions: bringing nfv to the network edge, IEEE Communications Magazine 55 (6) (2017) 24–31. doi:10.1109/MCOM.2017.1601039.

[21] M. Wajahat, Costefficient dynamic management of cloud resources through supervised learning, ACM SIGMETRICS Performance Evaluation Review 47 (3) (2020) 28–30.

[22] B. Li, W. Lu, S. Liu, Z. Zhu, Deep-learning-assisted network orchestration for on-demand and cost-effective vnf service chaining in inter-dc elastic optical networks, IEEE/OSA Journal of Optical Communications and Networking 10 (10) (2018) D29–D41.

[23] S. I. Kim, H. S. Kim, Method for vnf placement for service function chaining optimized in the nfv environment, in: 2019 Eleventh International Conference on Ubiquitous and Future Networks (ICUFN), IEEE, 2019, pp. 721–724.

[24] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, C. Delimitrou, Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices, in: Proceedings of

the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 19–33. doi:https://doi.org/10.1145/3297858.3304004.

[25] S. Van Rossem, W. Tavernier, D. Colle, M. Pickavet, P. Demeester, Optimized Sampling Strategies to Model the Performance of Virtualized Network Functions (Jan. 2020).
URL https://hal.archives-ouvertes.fr/hal-02354401

[26] Locust, an open source load testing tool., https://locust.io/, accessed: 2020-02-05.

[27] Flask, the python micro framework for building web applications., https://palletsprojects.com/p/flask/, accessed: 2020-02-05.

[28] Haproxy, the reliable, high performance tcp/http load balancer, http://www.haproxy.org/, accessed: 2020-02-05.

[29] Tinyproxy, lightweight http(s) daemon, https://tinyproxy.github.io/, accessed: 2020-02-05.

[30] pfsense, world's most trusted open source firewall, https://www.pfsense.org/, accessed: 2020-02-05.

[31] Open vswitch, production quality, multilayer open virtual switch, https://www.openvswitch.org/, accessed: 2020-02-05.

[32] M. Peuster, H. Karl, Understand your chains and keep your deadlines: Introducing time-constrained profiling for nfv, in: 2018 14th International Conference on Network and Service Management (CNSM), 2018, pp. 240–246.

[33] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, [Online; accessed 2019-04-23] (2001–).
URL http://www.scipy.org/

[34] C. K. Williams, C. E. Rasmussen, Gaussian processes for machine learning, Vol. 2, MIT press Cambridge, MA, 2006.
URL http://www.gaussianprocess.org/gpml/chapters/

[35] C. H. T. Arteaga, F. Rissoi, O. M. C. Rendon, An adaptive scaling mechanism for managing performance variations in network functions virtualization: A case study in an nfv-based epc, in: 2017 13th International Conference on Network and Service Management (CNSM), IEEE, 2017, pp. 1–7.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[37] D. P. Bovet, M. Cesati, Understanding the Linux Kernel: from I/O ports to process management, " O'Reilly Media, Inc.", 2005.

[38] M. Fisk, W.-c. Feng, Dynamic right-sizing in tcp, Tech. rep., Los Alamos National Lab., Los Alamos, NM (US) (2001).

[39] E. Weigle, W.-c. Feng, A comparison of tcp automatic tuning techniques for distributed computing, in: Proceedings 11th IEEE International Symposium on High Performance Distributed Computing, IEEE, 2002, pp. 265–272.

[40] D. Borman, et al., Rfc 7323: Tcp extensions for high performance.
URL https://tools.ietf.org/html/rfc7323

[41] J. Corbet, Jls2009: Generic receive offload, Linux Weekly News (LWN)Accessed: 2020-02-10.
URL https://lwn.net/Articles/358910/

[42] T. Herbert, W. de Bruijn, Scaling in the linux networking stack, URL https://www. kernel. org/doc/Documentation/networking/scaling. txtAccessed: 2020-02-05.

[43] E. Altman, D. Barman, B. Tuffin, M. Vojnovic, Parallel tcp sockets: Simple model, throughput and validation., in: INFOCOM, Vol. 2006, 2006, pp. 1–12.

[44] J. Lei, K. Suo, H. Lu, J. Rao, Tackling parallelization challenges of kernel network stack for container overlay networks, in: 11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19), 2019.

[45] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, A. Akella, Iron: Isolating network-based CPU in container environments, in: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), USENIX Association, Renton, WA, 2018, pp. 313–328.

[46] L. Chen, Curse of Dimensionality, Springer US, Boston, MA, 2009, pp. 545–546. doi:10.1007/978-0-387-39940-9_133.
URL https://doi.org/10.1007/978-0-387-39940-9_133

[47] C. Laaber, J. Scheuner, P. Leitner, Software microbenchmarking in the cloud. how bad is it really?, Empirical Software Engineering 24 (4) (2019) 2469–2508.

[48] D. Zhuo, Q. Zhang, X. Yang, V. Liu, Canaries in the network, in: Proceedings of the 15th ACM

Workshop on Hot Topics in Networks, 2016, pp. 36–42.
[49] A. Wilson, R. Adams, Gaussian process kernels for pattern discovery and extrapolation, in: International conference on machine learning, 2013, pp. 1067–1075.